

Query by Humming System

Amiya Kumar Tripathy¹, Neha Chhatre², Namrata Surendranath³ and Manpreet Kalsi⁴

Don Bosco Institute of Technology, Mumbai, India

¹tripathy.a@gmail.com, ²neha_chhatre@yahoo.co.in,

³namratasurendranath@yahoo.com, ⁴mann_k_0905@yahoo.co.in

Abstract— A Query by Humming system allows the user to find a song by humming part of the tune. No musical training is needed. The idea is simple: you hum into the microphone, the computer records the hum and extracts certain features corresponding to the melody and rhythm characteristics, and it then compares the features to the features of the songs in the database. Finally it returns a ranked list of the songs or song segments most similar to the humming. The goal is to build a reliable and efficient large-scale system that collects thousands of songs and responds in seconds.

Index Terms—Query Processing, Humming System, third term, fourth term, fifth term, sixth term

I. INTRODUCTION

Query by humming is an interaction concept in which the identity of a song has to be revealed fast and orderly from a given sung input using a large database of known melodies. In short, it tries to detect the pitches in a sung melody and compares these pitches with symbolic representations of the known melodies. Melodies that are similar to the sung pitches are retrieved. Approximate pattern matching in the melody comparison process compensates for the errors in the sung melody by using classical dynamic programming. A filtering method is used to save computation in the dynamic programming.

Although a substantial number of research projects have addressed music information retrieval over the past three decades, the field is still very immature. Few of these projects involve complex (polyphonic) music; methods for evaluation are at a very primitive stage of development; none of the projects tackles the problem of realistically large-scale databases. Many problems to be faced are due to the nature of music itself. Among these are issues in human perception and cognition of music, especially as they concern the recognizability of a musical phrase. Searching on pitch (or pitch-contour) is likely to be satisfactory for all purposes. This assumption may indeed be true for most monophonic (single-voice) music, but it is certainly inadequate for polyphonic (multi-voice) music. Even in the monophonic case it can lead to misleading results.

Query by humming (QBH) is a music retrieval system that branches of the original classification systems of title, artist, composer, and genre. The system involves

taking a user-hummed melody (input query) and comparing it to an existing database. The system then returns a ranked list of music closest to the input query. The interaction concept of query by humming makes it possible to retrieve a song when the user ponders a catchy tune without being able to name the song. It allows the user to sing any melodic passage of a song, while the system seeks the song containing that melody fast and orderly [1].

Next generation databases will include image, audio and video data in addition to traditional text and numerical data [9]. These data types will re-quire query methods that are more appropriate and natural to the type of respective data. For instance, a natural way to query an image database is to retrieve images based on operations on images or sketches supplied as input. Similarly a natural way of querying an audio database (of songs) is to hum the tune of a song. Such a system would be useful in any multimedia database containing musical data by providing an alternative and natural way of querying. One can also imagine a widespread use of such a system in commercial music industry, music radio and TV stations, music stores and even for one's personal use.

A melody retrieval system based on acoustic querying would allow a user to hum or sing a short fragment of a song into a microphone and then search and retrieve the best matched song from the database. Our project, a query-by-humming music indexing and retrieval system based on melody, or the tune, of the music. It deals with monophonic music where the basic method deployed is the conversion of input hum (wav file) to a digital representation (midi file format). It introduces the basic functional blocks and outlined the challenging problems posed by this application.

II. LITERATURE SURVEY

Tansen is a music retrieval system which is under development at IIT Bombay [1]. The system is designed to accept an acoustic query in the form of song fragments, to search a database of Indian film songs. TANSEN, a query-by humming music indexing and retrieval system based on melody, or the tune, of the music. It introduced the basic functional blocks during their starting phase. The melody database is essentially an indexed set of sound-tracks. The acoustic query, which is typically a

few notes whistled, hummed or sung by the user (presently restricted to the syllable *ta*), is processed to detect its melody line. The database is searched to find those songs that best match the query. The system returns a ranked set of matching melodies, which can be used to retrieve the desired original soundtrack.

The major algorithmic modules are the extraction of a melody representation from the query (and also the database songs at the time of creating the database), and the melodic similarity distance computation. While the overall task is one that is easily performed by humans, many challenging problems arise in the implementation of an automatic system. The system will typically operate on a substantial database and must respond within seconds. In this system, the first 20 notes of the query are considered. Dynamic programming is used for searching. Here 3-level pitch contour is used to represent melody. Whistling is the only form of querying supported. There exist several algorithms for detecting the pitch of an acoustic signal. We have used time domain autocorrelation function for pitch extraction since it is computationally simple and fast. It is computed on non-overlapping frames of fixed duration (equal to 3 times the lowest expected pitch period). The melody of a piece of music is a sequence of notes with varying pitch and duration. The pitch is associated with the periodicity of the sound, and allows the arranging of sounds ranked low to high on a musical scale. Although the melody is described by the time sequence of pitches, it is evident that People are able to recognize melodies even after pitch transposition (as the same tune played in a different key. The relative variation of pitch in time is known as the pitch contour, and it provides a dimension which is invariant to key transposition.

User queries cannot be expected to be completely accurate with respect to the actual pitch contour of the desired music. Typical inaccuracies are: (i) insertion of new notes (ii) replacement by different note (iii) deletion of notes. These inaccuracies can be taken care of by a dynamic programming (DP) based edit distance algorithm. Apart from this formal experiment, the system has been tested informally by a large number of people and has shown a high degree of robustness. Of immediate importance is increasing the number of songs in the database. This work is underway, and it is expected that a convincing demo on a realistic database will be present [1].

III. PROPOSED METHODOLOGY

The major algorithmic modules are the extraction of a melody representation from the query (and also the database songs at the time of creating the database), and the melodic similarity distance computation. Hence we divide the system in to two layers.

Layer 1:

In this layer the user hums the required query .The query is recorded and stored as a .WAV file. This file is then fed to a Speech Synthesis Software: PRAAT.PRAAT accepts the query and gives the Pitch Contour of the hummed query. Detailed Pitch Contour Analysis along with efficient use of algorithms helps in extracting the notes from the hummed query.

Layer 2:

In this layer a Database Schema consisting of different genres of MIDI files is stored. Correspondingly their Pitch Contour Analysis is done using PRAAT and the notes extracted are stored in various files. Thus, the notes extracted from Layer 1 are evaluated against Layer 2 using an appropriate Pattern Matching Algorithm. The system returns the best matched melody to the requested query.

IV. SYSTEM ARCHITECTURE

The architecture is illustrated in Figure 1. Operation of the system is straight-forward. Queries are hummed into a microphone, digitized, and fed into a pitch-tracking module (PRAAT) [11]. The result, a contour representation of the hummed melody, is fed into the query engine, which produces a ranked list of matching melodies. The database of melodies will be acquired by processing public domain MIDI songs, and is stored as a flat file database. Pitch tracking can be performed. Hummed queries may be recorded in a variety of formats. We will be experimenting with the standard 16-bit; 22.05 KHz WAV format on a Linux system. The query engine uses an approximate pattern matching algorithm, in order to tolerate humming errors. The melody database is essentially an indexed set of soundtracks. The acoustic query, which is typically a few notes hummed (presently restricted to 'ta' syllable) by the user, is processed to detect its melody line. The database is searched to find those songs that best match the query.

While the overall task is one that is easily performed by humans, many challenging problems arise in the implementation of an automatic system. These include the signal processing needed for extracting the melody from the stored audio and from the acoustic query, and the pattern matching algorithms to achieve proper ranked retrieval. Further, a robust system must be able to account for inaccuracies in the user's singing. The system will typically operate on a substantial database and must respond within seconds.

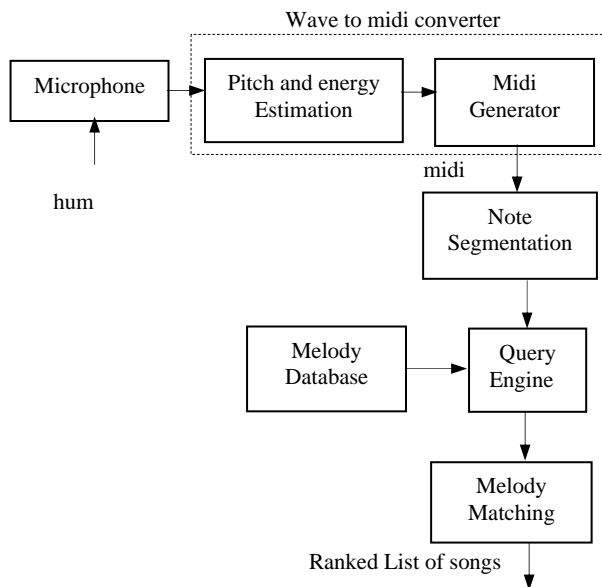


Fig 1: System Architecture

V. IMPLEMENTATION LOGIC

MIDI file format:

The Standard MIDI File SMF is a file format specifically designed to store the data that a sequencer records and plays (whether that sequencer be software or hardware based). This format stores the standard MIDI messages (i.e., status bytes with appropriate data bytes) plus a time-stamp for each message (i.e., a series of bytes that represent how many clock pulses to wait before playing the event). The format allows saving information about tempo, pulses per quarter note resolution (or resolution expressed in divisions per second, i.e. SMPTE setting), time and key signatures, and names of tracks and patterns. It can store multiple patterns and tracks so that any application can preserve these structures when loading the file. The format was designed to be generic so that any sequencer could read or write such a file without losing the most important data, and extensible enough for a particular application to store its own proprietary, extra data in such a way that another application won't be confused when loading the file and can safely ignore this extra stuff that it doesn't need. Think of the MIDI file format as a musical version of an ASCII text file (except that the MIDI file contains binary data too), and the various sequencer programs as text editors all capable of reading that file. But, unlike ASCII, MIDI file format saves data in chunks (i.e., groups of bytes preceded by an ID and size) which can be parsed, loaded, skipped, etc. Therefore, it can be easily extended to include a program's proprietary info.

For example, maybe a program wants to save a flag

byte that indicates whether the user has turned on an audible metronome click. The program can put this flag byte into a MIDI file in such a way that another application can skip this byte without having to understand what that byte is for. In the future, the MIDI file format can also be extended to include new official chunks that all sequencer programs may elect to load and use. This can be done without making old data files obsolete (i.e., the format is designed to be extensible in a backwardly compatible way). In conclusion, any software that saves or loads MIDI data should use SMF format for its data files. Standard MIDI files provide a common file format used by most musical software and hardware devices to store song information including the title, track names, and most importantly what instruments to use and the sequence of musical events, such as notes and instrument control information needed to play back the song.

This standardization allows one software package to create and save files that can later be loaded and edited by another completely different program, even on a different type of computer. Almost every software music sequencer is capable of loading and saving standard MIDI files. Data is always saved within a chunk. There can be many chunks inside of a MIDI file. Each chunk can be a different size (and likely will be). A chunk's size is how many (8-bit) bytes are contained in the chunk. The data bytes in a chunk are typically related in some way. For example, all of the bytes in one chunk may be for one particular sequencer track. The bytes for another sequencer track may be put in a different chunk. So, a chunk is simply a group of related byte. Each chunk must begin with a 4 character (i.e., 4 ASCII bytes) ID which tells what type of chunk this is. The next 4 bytes must form a 32-bit length (i.e., size) of the chunk. All chunks must begin with these two fields (i.e., 8 bytes), which are referred to as the chunk header [13].

Here's what a chunk's header looks like if you defined it in C:

```

Struct CHUNK HEADER
{
    Char ID[4];
    Unsigned long Length;
};
    
```

The Length does not include the 8 byte chunk header. It simply tells you how many bytes of data are in the chunk following this header and here's an example chunk header (with bytes expressed in hex) if you examined it with a hex editor:

```
4D 54 68 64 00 00 00 06
```

Note that the first 4 bytes make up the ASCII ID of MThd (i.e., the first four bytes are the ASCII values for 'M', 'T', 'h', and 'd'). The next 4 bytes tell us that there

should be 6 more data bytes in the chunk (and after that we should find the next chunk header or the end of the file). The 4 bytes that make up the Length are stored in (Motorola) Big Endian byte order, not (Intel) Little Endian reverses byte order. (i.e., The 06 is the fourth byte instead of the first of the four). In fact, all MIDI files begin with the above MThd header (and that's how you know that it's a MIDI file). The MThd header has an ID of MThd, and a Length of 6. The first two data bytes tell the Format. There are actually 3 different types (i.e., formats) of MIDI files. A type of 0 means that the file contains one single track containing midi data on possibly all 16 midi channels.

If your sequencer sorts/stores all of its midi data in one single block of memory with the data in the order that it's played, then it should read/write this type. A type of 1 means that the file contains one or more simultaneous (i.e., all start from an assumed time of 0) tracks, perhaps each on a single midi channel. Together, all of these tracks are considered one sequence or pattern. If your sequencer separates its midi data (i.e. tracks) into different blocks of memory but plays them back simultaneously (i.e., as one pattern), it will read/write this type. A type of 2 means that the file contains one or more sequentially independent single-track patterns. If your sequencer separates its midi data into different blocks of memory, but plays only one block at a time (i.e., each block is considered a different excerpt or song), then it will read/write this type. The next 2 bytes tell how many tracks are stored in the file, NumTracks. Of course, for format type 0, this is always 1. For the other 2 types, there can be numerous tracks. The last two bytes indicate how many Pulses (i.e. clocks) Per Quarter Note (abbreviated as PPQN) resolution the time-stamps are based upon, Division. For example, if your sequencer has 96 ppqn, this field would be (in hex):

```
00 60
```

The 2 bytes that make up the Division are stored in (Motorola) Big Endian byte order, not (Intel) Little Endian reverses byte order. The same is true for the NumTracks and Format. Alternately, if the first byte of Division is negative, then this represents the division of a second that the timestamps are based upon. The first byte will be -24, -25, -29, or -30, corresponding to the 4 SMPTE standards representing frames per second. The second byte (a positive number) is the resolution within a frame (i.e., sub frame). Typical values may be 4 (MIDI Time Code), 8, 10, 80 (SMPTE bit resolution), or 100. Here's what an MThd chunk looks like if you defined it in C:

```
Struct MTHD_CHUNK
{
/* Here's the 8 byte header that all chunks must have*/

Char ID[4]; /* this will be 'M', 'T', 'h', 'd' */

Unsigned long Length; /*this will be 6*/
```

```
/* Here are the 6 bytes*/
Unsigned short Format;
Unsigned short NumTracks;
```

```
Unsigned short Division;
```

And here's an example of a complete MThd chunk (with header) if you examined it in a hex editor:

```
4D 54 68 64    MThd id
00 00 00 06    Length of the MThd chunk is always 6
00 01          the format type is 1
00 02          there are 2 MTrk chunks in the file
```

```
E7 28
```

each increment of delta-time represents a millisecond. After the MThd chunk, one will find an, MTrk chunk as this is the only other currently defined chunk. An MTrk chunk contains all of the midi data (with timing bytes), plus optional non-midi data for one track. Obviously, you should encounter as many MTrk chunks in the file as the MThd chunk's NumTracks field indicated. The MTrk header begins with the ID of MTrk, followed by the Length (i.e., number of data bytes for this track). The Length will likely be different for each track.

Here's what an MTrk chunk looks like if you defined it in C:

```
Struct MTRK_CHUNK
{
/* Here's the 8 byte header that all chunks must have*/

Char ID [4]; /* this will be 'M', 'T', 'r', 'k'*/

Unsigned long Length; /* this will be the actual size of
Data [ ] */

/* Here are the data bytes */

Unsigned char Data [ ]; /* its actual size is Data
[Length] */

};
```

WAV file format:

The Wave file format is Windows' native file format for storing digital audio data. It has become one of the most widely supported digital audio file formats on the PC due to the popularity of Windows and the huge number of programs written for the platform. Almost every modern program that can open and/or save digital audio supports this file format, making it both extremely useful and a virtual requirement for software developers to understand. It supports a variety of bit resolutions,

sample rates, and channels of audio. This format is very popular upon IBM PC (clone) platforms, and is widely used in professional programs that process digital audio waveforms. It takes into account some peculiarities of the Intel CPU such as little Endian byte order. This format uses Microsoft's version of the Electronic Arts Interchange File Format method for storing data in chunks.

The WAVE format is a subset of RIFF used for storing digital audio. Its form type is WAVE, and it requires two kinds of chunks: the fmt chunk, which describes the sample rate, sample width, etc., and the data chunk, which contains the actual samples. WAVE can also contain any other chunk type allowed by RIFF, including LIST chunks, which are used to contain optional kinds of data such as the copyright date, author's name, etc. Chunks can appear in any order. The WAVE file is thus very powerful, but also not trivial to parse. This subset basically consists of only two chunks, the fmt and data chunks, in that order, with the sample data in PCM format. The WAVE specification supports a number of different compression algorithms. The format tag entry in the fmt chunk indicates the type of compression used. A value of 1 indicates Pulse Code Modulation (PCM), which is a straight uncompressed encoding of the samples. Values other than 1 indicate some form of compression.

WAV to MIDI:

To create a MIDI a file for a song recorded in WAV format a musician must determine pitch, velocity and duration of each note being played and record these parameters into a sequence of MIDI events. The Midi created represents the basic melody and chords of recognized music. The difference between WAV and MIDI formats consists in representation of sound and music. WAV format is digital recording of any sound (including speech) and MIDI format is principally sequence of notes (or MIDI events). Here we have an Output File (.mid) from an Input File (.wav) that contains musical data, and a Tone File (.wav) that consists of monotone data. An advantage of such a structure is also the fact that the query is prepared on the client side of the system. In this case the query is very short. Besides, there is a possibility to evaluate its quality before sending to the server. The system provides for playback of the recognized melody notes in MIDI format. This allows the user to listen to a query and take a decision either to send it to the server or to sing it once again.

Algorithm for WAV to MIDI:

Part 1

1. Accept the user query and save it as a .wav file
2. Perform pitch energy estimation on the wav files.
3. From the respective pitch tier calculate the frequencies of various notes at different time intervals.
4. Using the formula

$$p = 69 + 12 \times \log_2(f \div 440)$$

Part 2: Generation of midi file

1. Declare the data structures for mthd header and the mtrk header according to the midi file format.
2. For the data bytes to be written into the midi file, each event will consist of two events time and message of which the time also called as delta time is a variable length quantity.
3. Delta time can be written using the following pseudo code.
 1. unsigned long result
 2. unsigned long array [4]
 3. count=0
 4. if (result less than 128)
 - goto step 9
 5. array [count] =((result & 0x7F) | 0x80)
 6. count++;
 7. shift result by 7 bits right
 8. goto step 3
 9. array [count]=(result & 0x7F)
 10. count++
 11. for (i=count to 0)
 - display array in reverse order.

VI. MELODY REPRESENTATION AND PROCESSING QUERY

The fundamental attributes of music are the pitch sequence of notes, rhythm, tempo (slow/fast), dynamics (loud/soft), texture (timbre or voices) and lyrics (if any). It is in these dimensions that we typically distinguish one piece of music from another. Of these descriptors, melody and rhythm are the most distinctive. The melody of a piece of music is a sequence of notes with varying pitch and duration.

Reliable note segmentation is a critical aspect of query processing. In order to simplify note segmentation, we currently require the query we sung using syllable 'ta'. The stop consonant's' causes the local energy of the waveform to dip thus making for relatively easy identification of note boundaries. We compute the instantaneous energy of query waveform averaged over 25 ms frames.

VII. STRING MATCHING

The database is a set of songs indexed by the melody string of the signature phrase (or the most easily recalled phrase) of the song. Extracting the melody representation from the original soundtrack is a difficult problem. Currently, we obtain *model* queries from a trained singer

and use these to obtain the melody representation for the database songs. User queries cannot be expected to be completely accurate with respect to the actual pitch contour of the desired music.

Typical inaccuracies are:

1. Insertion of new notes
2. Replacement by different note
3. Deletion of notes: These inaccuracies can be taken care of by a Dynamic Programming (DP) based edit distance algorithm.

VIII. HUM TO STRING CONVERSION

User input to the system (humming) is converted into a sequence of relative pitch transitions. A note in the input is classified in one of three ways: a note is either the same as the previous note (S), higher than previous note (U), or lowers than the previous note (D). Thus, the input is converted into a string with a three letter alphabet (U, D, S).

For example consider a SONG has notes as follows:
48 52 52 47 55 58

This is converted to a U, D, S string as follows:
USDUU

IX. PATTERN MATCHING

The database is a set of songs indexed by the melody string of the signature phrase (or the most easily recalled phrase) of the song. Extracting the melody representation from the original soundtrack is a difficult problem. There are many algorithms possible for pattern matching. Here we do pattern matching in U, D, S form. We convert the wav file into U, D, S strings and the database where our songs are stored is also converted into U, D, S strings. These strings are then compared using the edit distance algorithm. The edit distance [20] of two strings, s1 and s2, is defined as the minimum number of point mutation required to change s1 into s2, where a point mutation is one of:

1. Change a letter, 2. Insert a letter or 3. Delete a letter.
- This algorithm averages pitch values within the 50 to 80 percentage of the duration of the note.

$$d_{ij} = \min \begin{cases} d_{i-1, j} + w(a_i, 0) & \text{(deletion)} \\ d_{i-1, j-1} + w(a_i, b_j) & \text{(match/change)} \\ d_{i, j-1} + w(0, b_j) & \text{(insertion)} \end{cases}$$

The initial conditions are:

$$\begin{aligned} d_{0,0} &= 0 \\ d_{i,0} &= d_{i-1,0} + w(a_i, 0), i \geq 1 \\ d_{0,j} &= d_{0,j-1} + w(0, b_j), j \geq 1 \end{aligned}$$

Where $w(a_i, 0)$ is the weight associated with the deletion of a_i , $w(0, b_j)$ is the weight for insertion of b_j , and $w(a_i,$

$b_j)$ is the weight for replacement of element i of sequence A by element j of sequence B. The operation titled "match/change" sets $w(a_i, b_j) = 0$ if $a_i = b_j$ and a value greater than 0 if $a_i \neq b_j$. The weights used here are 1 for insertion, deletion and substitution(change) and 0 for match. As an example, if two pitch contour strings *UDDSSUD and *UDDSSUD are compared, the edit distance is 1.

X. RESULTS AND CONCLUSION

For the testing we have used records of melodies fragments of popular compositions of nursery rhymes, few hit film songs and nursery rhymes performed. Records format was: PCM 22.05 kHz, 16 bit, mono, normalized by volume. We used melodies of three types: a) melodies hummed clearly, with out distortions; b) melodies hummed falsely. There was only one requirement - to sing using the syllables [ta] or [da]. No other requirements such as compliance with musical key or tempo for a melody performance were made.

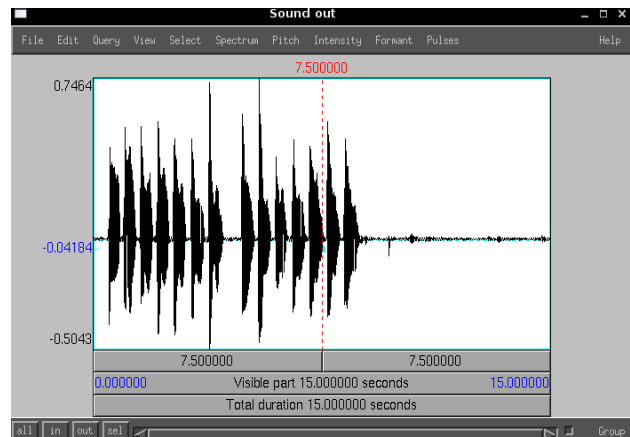


Fig 2: Melody of a perfect hum

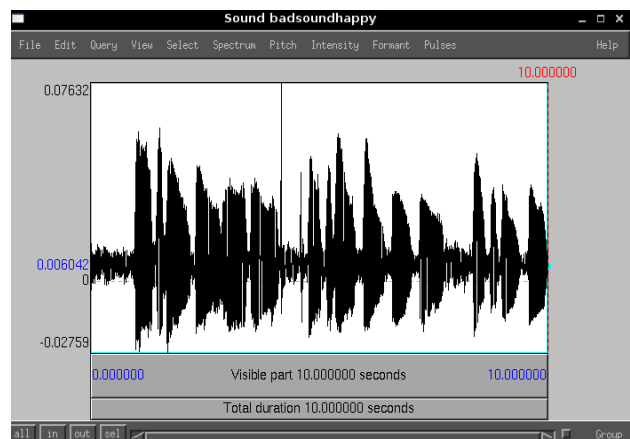


Fig 3: Melody of a bad hum

Our evaluation tested the tolerance of the system with respect to input errors, whether from mistakes in the user's humming or from problems with the pitch-tracking. The effectiveness of this method is directly related to the accuracy with which pitches that are

hummed can be tracked and the accuracy of the melodic information within the database.

Under ideal circumstances, we can achieve close to 100 percentage accuracy tracking humming, where ideal circumstances mean the user places a small amount of space between each note and hits each note strongly. For this purpose, humming short notes is encouraged. It is decided that the new system would have the following application:

Music search engine at music store, Ring tone search service for cell phones, Karaoke scoring, Music learning, etc.

Stand-alone system:

The whole processing takes place on one computer, e.g. for record stores or home use.

Web application:

The transmission of the input signal is done by a Java-Applet on a web site. All processing occurs on a remote server and the result is displayed on the user's web browser. Content providers can use the result list to obtain further information or offers (e.g. downloads, purchases).

Mobile application:

The user calls a remote server with his cell phone and sings the query. Again all processing is done on the server, which returns the result via short message service back to the user.

- [13] <http://www.borg.com/~jglatt/tech/miditech.htm>. (Accessed on 14 August 2007)
- [14] <http://www.midi.org/about-midi/abtmidi.shtml>. (Accessed on 15 September 2008)
- [15] <http://www.borg.com/~jglatt/tech/wave.htm>. (Accessed on 13 October 2007)
- [16] <http://www.sonicspot.com/guide/wavefiles.html>. (Accessed on 05 February 2008)
- [17] [http://en.wikipedia.org/wiki/Sine wave](http://en.wikipedia.org/wiki/Sine_wave). (Accessed on 06 February 2008)
- [18] <http://www.borg.com/~jglatt/tech/abouttiff.htm>. (Accessed on 06 February 2008)
- [19] <http://www.merriampark.com/ld.htm>. (Accessed on 23 September 2007)
- [20] <http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamics/Edit/> (Accessed on 05 February 2008)

REFERENCES

- [1] M. A. Raju, B. Sundaram, and P. Rao, TANSEN: A Query-By- Humming based Music Retrieval System, In Proc. National Conference on Communications (NCC), 2003.
- [2] Alan V. Oppenheim and Ronald W. Schaffer, Discrete-time Signal Processing, Prentice Hall, Fourth Edition.
- [3] Yashwant Kanetkar, Let Us C, BPB Publication, Fourth Edition.
- [4] Brian W. Kernighan and Dennis Ritchie, 'The ANSI C programming language', Tata McGraw-Hill, Fourth Edition.
- [5] Edmond Lau, Annie Ding, Calvin On, MUSICDB: A Query By humming system
- [6] Sumantra Dutta Roy, Preeti Rao, Contour based melody representation: An analytical study.
- [7] John G. Proakis, Digital Signaling and Processing, Prentice Hall, Fourth Edition.
- [8] [http://en.wikipedia.org/wiki/Query by humming](http://en.wikipedia.org/wiki/Query_by_humming). (Accessed on 11 September 2007)
- [9] <http://www.doc.ic.ac.uk/~srueger/pr-a.tarter-003/index.html>. (Accessed on 10 January 2008)
- [10] <http://www.cs.cornell.edu/Info/Faculty/bsmith/query-by-humming.html>. (Accessed on 13 January 2008)
- [11] <http://www.praat.org>. (Accessed on 14 January 2008)
- [12] [http://www.sloud.com/download/Sloud QBH Search Music. PDF](http://www.sloud.com/download/Sloud_QBH_Search_Music.PDF). (Accessed on 05 February 2008)